Using Learning Techniques in Invariant Inference

Alex Aiken
Aditya Nori
Rahul Sharma
Saurabh Gupta
Bharath Hariharan

Invariant Inference

- An old problem

- A different approach with two ideas:
  1. Separate invariant inference from the rest of the verification problem

Why?

```
                    Pre
for (B)             I
 {

  …  code …

}                   Post
```

Pre)I

I Æ B

{ code }

I

I Æ:B)

Post

Invariant Inference

- An old problem

- A different approach with two ideas:
  1. Separate invariant inference from the rest of the verification problem

  2. Guess the invariant from executions

Why?

- Complementary to static analysis
  - underapproximations
  - “see through” hard analysis problems
    - functionality may be simpler than the code

- Possible to generate many, many tests

Nothing New Under the Sun

- Sounds like DAIKON?
  - Yes!

- Hypothesize (many) invariants
  - Run the program
  - Discard candidate invariants that are falsified
  - Attempt to verify the remaining candidates

A Simple Program

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Instrument loop head
- Collect state of program variables on each iteration

A DAIKON-Like Approach

```
s = 0;
y = 0;
while( * )
{
   print(s,y);
   s := s + 1;
   y := y + 1;
}
```

- Hypothesize
  - s = y
  - s = 2y

- Data

s	y
0	0

A DAIKON-Like Approach

```
s = 0;
y = 0;
while( * )
{
   print(s,y);
   s := s + 1;
   y := y + 1;
}
```

- Hypothesize
  - s = y
  - ~~s = 2y~~
- Data

s	y
0	0
1	1

A DAIKON-Like Approach

```
s = 0;
y = 0;
while( * )
{
   print(s,y);
   s := s + 1;
   y := y + 1;
}
```

- Hypothesize
  - s = y
  - ~~s = 2y~~

- Data

s	y
0	0
1	1
2	2
3	3

Another Approach

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Data

s	y
0	0
1	1
2	2
3	3

Arbitrary Linear Invariant

$as + by = 0$

- Data

s	y
0	0
1	1
2	2
3	3

Observation

`as + by = 0`

s	y
0	0
1	1
2	2
3	3

$$\begin{array}{c} w \\ \hline a \\ b \end{array} = \begin{array}{c} 0 \\ 0 \end{array}$$

Observation

$$as + by = 0$$

$$\{ w \mid Mw = 0 \}$$

s	y
0	0
1	1
2	2
3	3

$$\begin{array}{c} w \\ \hline a \\ b \end{array} = \begin{array}{c} 0 \\ 0 \end{array}$$

Observation

$as + by = 0$

NullSpace(M)

s	y
0	0
1	1
2	2
3	3

$$\begin{array}{c} w \\ \hline a \\ b \end{array} = \begin{array}{c} 0 \\ 0 \end{array}$$

Linear Invariants

- Construct matrix M of observations of all program variables
- Compute NullSpace(M)
- All invariants are in the null space

Spurious "Invariants"

- All invariants are in the null space
  - But not all vectors in the null space are invariants

- Consider the matrix

s	y
0	0

- Need a check phase
  - Verify the candidate is in fact an invariant

An Algorithm

- Check candidate invariant
  - If an invariant, done
  - If not an invariant, get counterexample
    - A reachable assignment of program variables falsifying the candidate
- Add new row to matrix
  - And repeat

Termination

- How many times can the solve & verify loop repeat?
- Each counterexample is linearly independent of previous entries in the matrix
- So at most N iterations
  - Where N is the number of columns
  - Upper bound on steps to reach a full rank matrix

Summary

- Superset of all linear invariants can be obtained by a standard matrix calculation

- Counter-example driven improvements to eliminate all but the true invariants
  - Guaranteed to terminate

What About Non-Linear Invariants?

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + y;
  y := y + 1;
}
```

Idea

- Collect data as before
- But add more columns to the matrix
  - For derived quantities
  - For example, y^2 and s^2
- How to limit the number of columns?
  - All monomials up to a chosen degree d

[Nguyen, Kapur, Weimer, Forrest 2012]

What About Non-Linear Invariants?

```
s = 0;
y = 0;
while( * )
{
   print(s,y);
   s := s + y;
   y := y + 1;
}
```

1	s	y	s^2	y^2	sy
1	0	0	0	0	0
1	1	1	1	1	1
1	3	2	9	4	6
1	6	3	36	9	18
1	10	4	100	16	40

Solve for the Null Space

$$a + bs + cy + ds^2 + ey^2 + fsy = 0$$

$$\begin{array}{cccccc} 1 & s & y & s^2 & y^2 & sy \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 9 & 4 & 6 \\ 1 & 6 & 3 & 36 & 9 & 18 \\ 1 & 10 & 4 & 100 & 16 & 40 \end{array} \quad \begin{array}{c} w \\ \hline a \\ b \\ c \\ d \\ e \\ f \end{array} \quad = \quad \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}$$

Candidate invariant: $-2s + y + y^2 = 0$

Comments

- Same issues as before
  - Must check candidate is implied by precondition, is inductive, and implies the postcondition on termination
  - Termination of invariant inference guaranteed if the verifier can generate counterexamples
- Experience: Solvers do well as checkers!

Experiments

Name	#vars	deg	Data	#and	Guess time (sec)	Check time (sec)	Total time (sec)
Mul2	4	2	75	1	0.0007	0.010	0.0107
LCM/GCD	6	2	329	1	0.004	0.012	0.016
Div	6	2	343	3	0.454	0.134	0.588
Bezout	8	2	362	5	0.765	0.149	0.914
Factor	5	3	100	1	0.002	0.010	0.012
Prod	5	2	84	1	0.0007	0.011	0.0117
Petter	2	6	10	1	0.0003	0.012	0.0123
Dijkstra	6	2	362	1	0.003	0.015	0.018
Cubes	4	3	31	10	0.014	0.062	0.076
geoReihe1	3	2	25	1	0.0003	0.010	0.0103
geoReihe2	3	2	25	1	0.0004	0.017	0.0174
geoReihe3	4	3	125	1	0.001	0.010	0.011
potSumm1	2	1	5	1	0.0002	0.011	0.0112
potSumm2	2	2	5	1	0.0002	0.009	0.0092
potSumm3	2	3	5	1	0.0002	0.012	0.0122
potSumm4	2	4	10	1	0.0002	0.010	0.0102

Summary to This Point

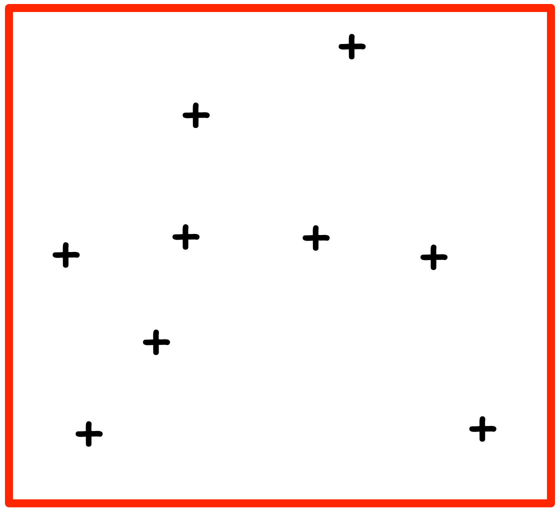
- Sound and complete algorithm for algebraic invariants
  - Up to a given degree

- Guess and Check
  - Hard part is inference done by matrix solve
  - Check part done by standard SMT solver
  - Much simpler and faster than previous approaches

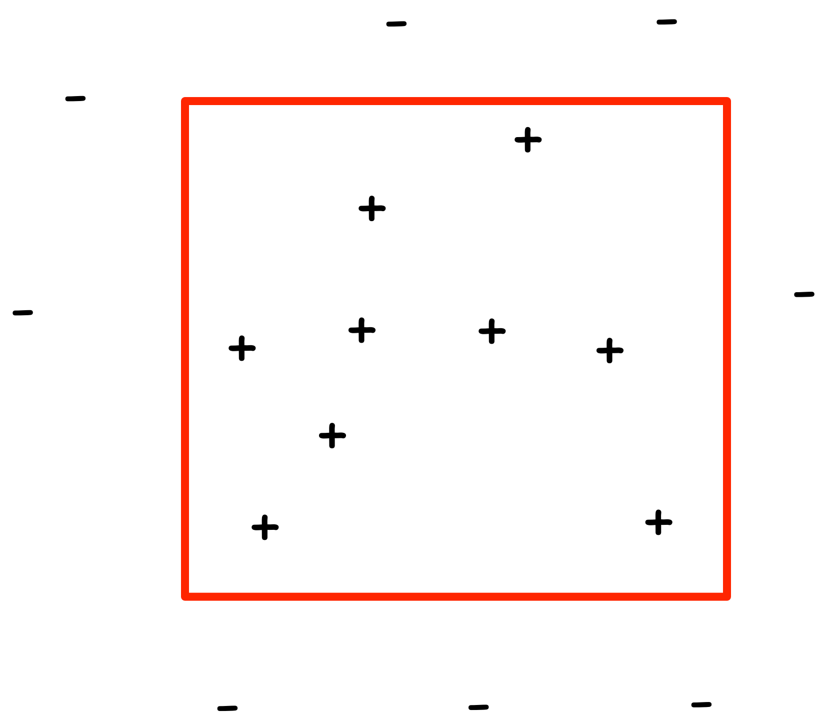
What About Disjunctive Invariants?

- Disjunctions are expensive
- Existing techniques severely restrict disjunctions
  - E.g., to a template

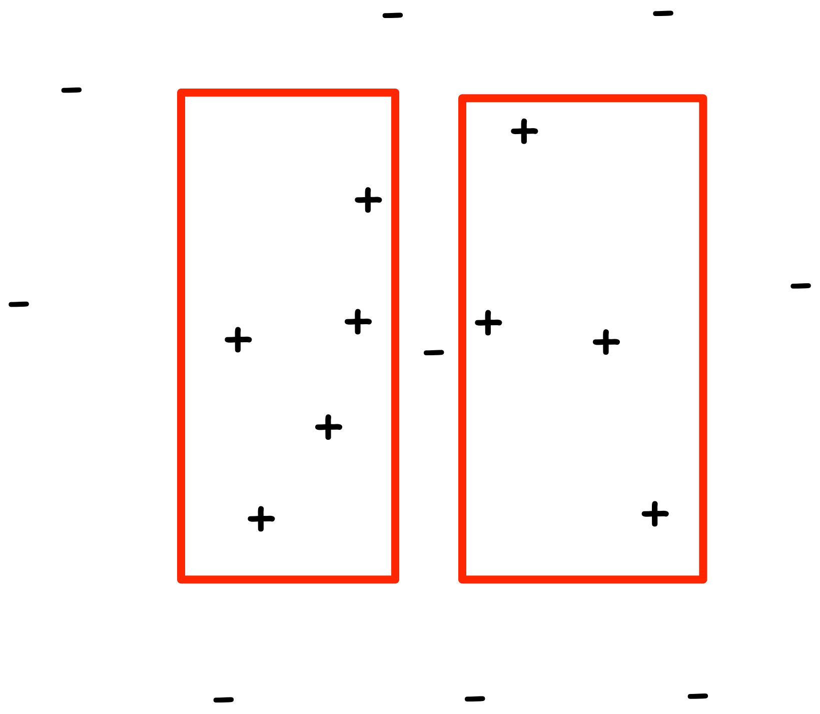
Good States



Separating Good States and Bad States



Separating Good States and Bad States



More Precisely . . .

- A *state* is a valuation of program variables

- Correct programs have good and bad states
  - All reachable states are good
    - Because we assume the program is correct
  - Assertions define the bad states
    - States that would result in the assertion being violated

- An invariant is a separator
  - Of the good states from the bad states

From Verification to Machine Learning

- From data we want to learn a separator of the good and bad states

- This is a machine learning problem

Goals

- Produce boolean combination of linear inequalities
  - Without templates
- Predictive
  - Generalizes well from small test suite
- Efficient
  - Hard, but more on this later

PAC Learning

- Given some positive and negative examples
  - Learn separator

- Separator is Probably Approximately Correct
  - With confidence 1 – x the accuracy is 1 – e
  - The number of examples is m = poly(1/x,1/e,d)

Example for Good and Bad States

```
x := y;
while(x != 0) do
  x := x-1;
  y := y-1;
assert y = 0
```

- Good states:
  - (x,y)=(1,1), (2,2),...
- Bad states:
  - SAT($x=0 \wedge y \neq 0$)
  - SAT($x=1 \wedge y \neq 1$)

Invariants

- Arbitrary boolean combination of
  - Equalities and
  - Inequalities
  - Over program quantities

- Note “program quantities” includes variables and induced quantities (like x^2)

First Part

- Run tests to get good states
- Run previous algorithm to infer equalities E
- Sample bad states
  - Consider while B do S; assert Q
  - Sample from :B Æ :Q Æ E
  - Sample from :B Æ WP(assume(B);S,:Q) Æ E

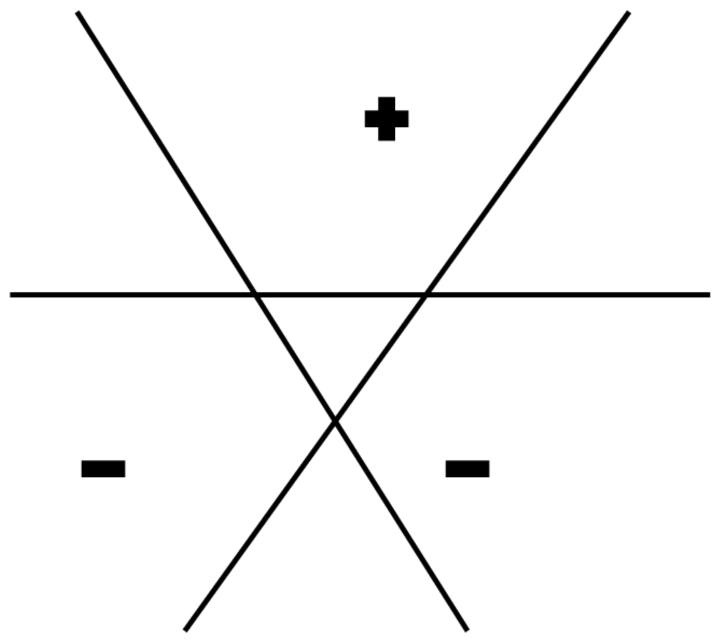
Idea

- Good and bad states are points in d-dimensional space
- Inequalities are planes in this space
- Must pick a set of planes that separate every good from every bad state

Picture

- How many planes are required?
  - At most m^d
  - m is # points
  - d is dimensionality

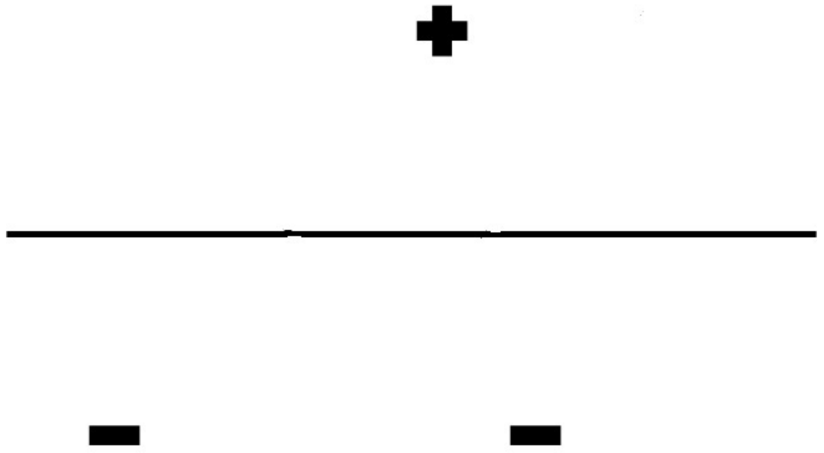
- Puts every point in its own cell



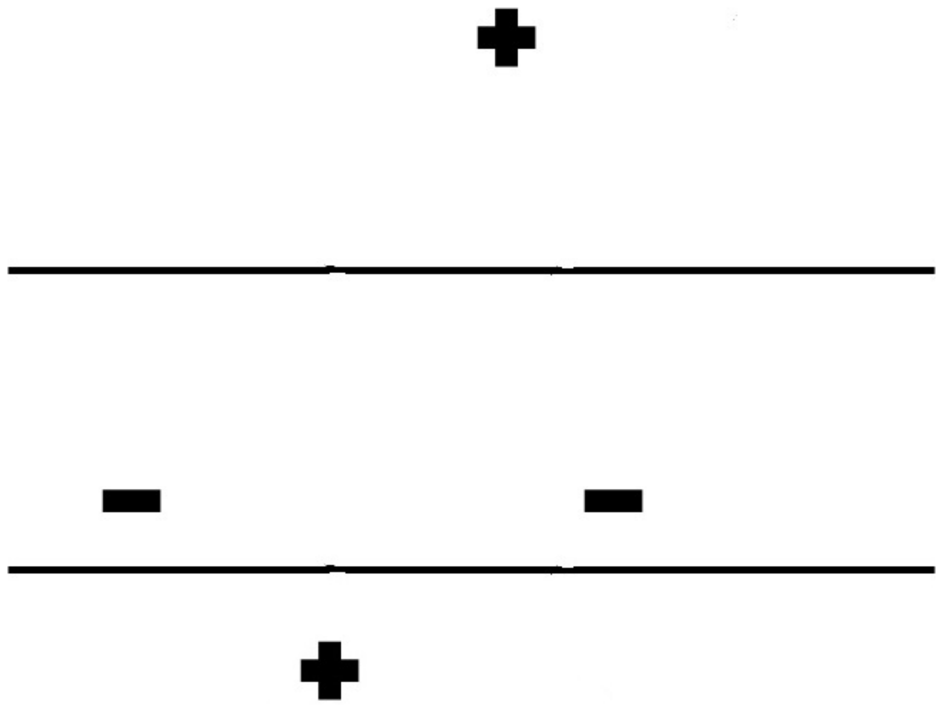
Theorem

- m^d planes (inequalities) would be awful
- PAC learning can find a subset of the planes that separate the positive and negative points
  - With $O(s \log m)$ planes
  - Where s is the size of the minimal separator
  - And m is roughly $ds \log ds$ …(other factors) …
  - In time m^{d+2}

Simple Example



Disjunction Example



Algorithm

- Consider a bipartite graph
  - Connects every good and bad state
- Repeat
  - Pick a plane cutting the maximum number of remaining edges

Analysis Ingredients

- m^d possible planes
- $s = m^2$ are a separator
- The greedy strategy in time m^{d+2} finds $s \log m$ planes

Comments

- The fact that there is only a log factor increase in number of planes over the minimum is important
  - Avoids overfitting

- In practice, the number of planes is small

Efficiency

- The general algorithm is too inefficient
- Impose some assumptions common to verification techniques
  - Reduce set of candidate planes to polynomial

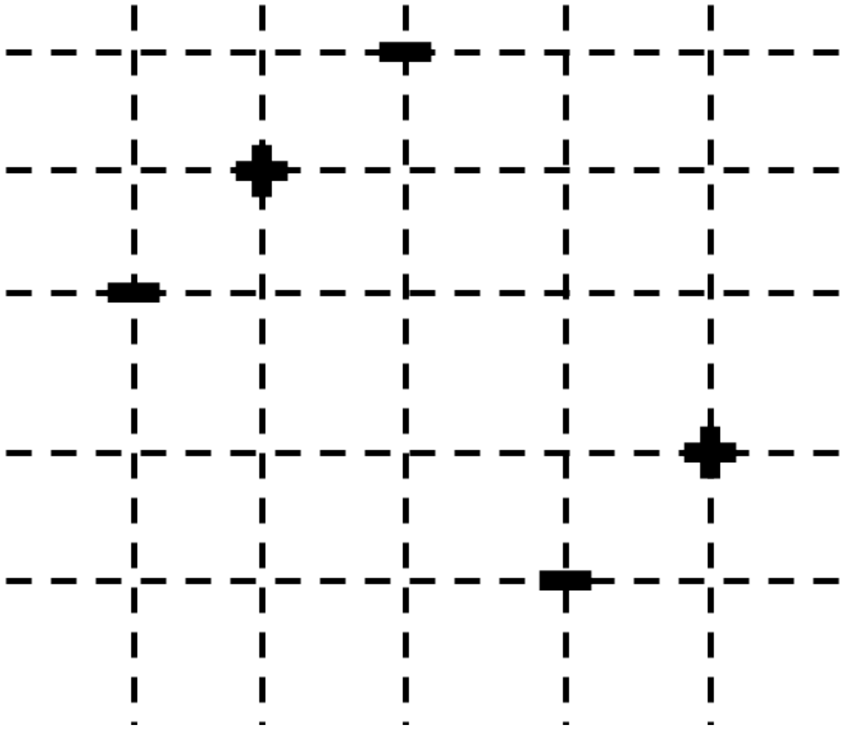
Predicate Abstraction

- The invariant is an (arbitrary boolean combination) of predicates in T
- Can find a PAC separator in time $O(m^2|T|)$
  - Even though the complexity of finding an invariant is NP^{NP} complete

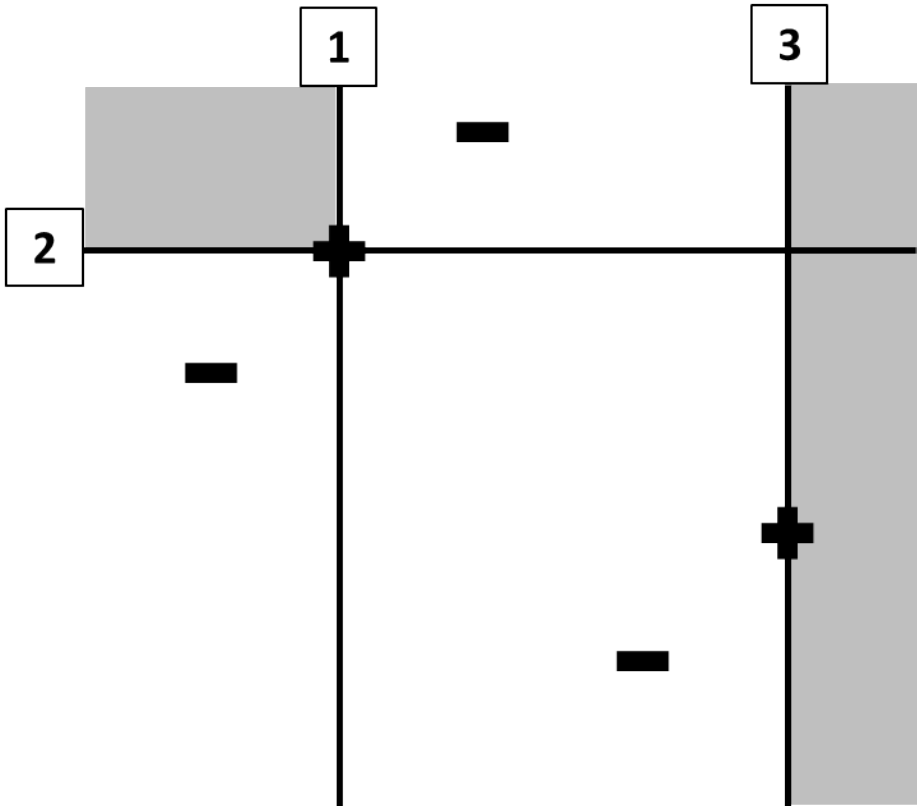
Abstract Interpretation

- Efficient algorithms for restricted abstract domains
  - Boxes $O(m^3d)$
  - Octagons $O(m^3d^2)$

Boxes



Boxes



Check Phase

- Use Boogie

- For counter-examples
  - Satisfies precondition, add as positive example
  - Violates assertion, add as negative example
  - If can't label, add as a constraint
    - Increases the guess size

Experiments

hsort	47	2	5	0.19	1.05	OK
msort	73	6	10	0.093	1.12	OK
nested	21	3	4	0.24	0.99	OK
seq-len1	44	6	5	4.39	1.04	PRE
seq-len	44	6	5	0.32	1.04	OK
svd	50	5	5	4.92	0.99	OK
esc-abs	71	2	6	1.09	1.06	OK
get-tag	120	2	2	0.092	1.04	OK
maill-qp	92	1	3	0.11	1.05	OK
spam	57	2	5	1.01	1.05	OK
split	20	1	5	FAIL	NA	FAIL
div	28	1	6	2.03	TO	OK

Application: Equality Checking

- Have extended these techniques to checking equality of arbitrary loops
  - Guess and verify a simulation relation
  - Mine equalities between the two loops as a guide

- Able to prove code generated by gcc –O2 equivalent to CompCert

Discussion

- Sound invariant inference based on PAC learning
- Machine learning/data mining techniques to
  - Handle disjunctions
  - Non-linearities
- Connects complexity of learning and complexity of verification

Discussion

- Like predecessors, focus on numerical invariants
  - Many other interesting aspects of programs not covered
  - Data structures, arrays, concurrency, higher-order functions …

- This is where we are headed …

Thanks!

Questions?